Introduzione

sabato 7 dicembre 2019 12.37

Deployment La fase in cui decidiamo come allocare le risorse logiche alle risorse fisiche disponibili.
Protocollo Complesso di regole e procedure cui ci si deve attenere in determinate attività per la esecuzione corretta

Modello cliente/servitore

Il modello standard che considereremo durante il corso è uno a molti: 1 server e N clienti e ha le seguenti proprietà:

- **Sincrono** si prevede risposta dal servitore al cliente
- **Bloccante** il cliente aspetta la risposta dal servitore
- **Asimmetrico** il cliente conosce il servitore ma il servitore non conosce a priori i clienti possibili
- **Dinamico** il binding tra cliente e servitore è dinamico, ossia il servitore che risponde alle richieste può cambiare tra diverse invocazioni

Modelli di interazione

Modello polling il cliente invia una sequenza di richieste al server ad intervalli di tempo a sua discrezione finchè non otttiene risposta
Modello push prima si stabilisce un rapporto C/S dal cliente al servitore a cui segue un rapporto C/S dal servitore al cliente (l'iniziativa passa al servitore). Un esempio sono i feed RSS (RDF Site Summary)

Entrambi i casi descritti sopra fanno riferimento a un modello sincrono bloccante in cui il cliente attende la ricezione della risposta senza poter proseguire la sua esecuzione. Esistono anche modelli non bloccanti in cui il cliente può eseguire altre operazioni nell'attesa della risposta dal server. In caso di interazione sincrona non bloccante si possono delegare le funzionalità di ricezione del risultato ad una entità che opera al posto del responsabile. Tali entità sono dette **proxy**, delegate, agenti, attori.

Modelli di imterconnessione

Interconnessione logica:

- connection oriented - TCP - affidabile e preserva l'ordine di invio dei messaggi
- connectionless - UDP - non affidabile non preserva l'ordine di invio di messaggi

Interconnessione fisica:

- connessione (OSI) - tutti i messaggi seguono la stessa strada
- senza connessione (IP) - i messaggi possono seguire strade diverse con strategia **best effort** di consegna pacchetti

Alcuni modelli a connessione (**TCP basato su IP**), non impegnano risorse intermedie ma solo sul mittente / destinatario

Stato nel modello cliente/servitore

Nella interazione C / S, un aspetto centrale è lo stato della interazione ossia che si tenga traccia della comunicazione e delle azioni precedenti. Il modello C/S può dunque essere:

- **stateless** - non si tiene traccia dello stato: ogni messaggio è completamente indipendente dagli altri
- **stateful** - si mantiene lo stato dell'interazione tra chi interagisce: un messaggio (e operazione conseguente) può dipendere da quelli precedenti

In genere, il cliente richiedente tende a livello applicativo a tenere traccia dello stato anche in meccanismo stateless.
Il modello di server stateless è più leggero e affidabile in presenza di malfunzionamenti ma lo stato deve essere mantenuto da ogni cliente.

Nel modello stateful invece lo stato è tipicamente mantenuto dal server. Se lo stato è presente, l'interazione è facilitata essendo riconosciuta e gestita in termini di protocollo (maggiore efficienza) ma abbiamo una minore affidabilità in presenza di malfunzionamenti.
Es. efficienza di un server stateful nella lettura di un elemento di un array di 10 elementi:
Usando lo stato ogni richiesta non deve specificare l'indice da chiedere, ma solo la nuova esigenza del dato successivo

In genere un'interazione stateless è sensata e possibile solo se il protocollo è progettato per operazioni **idempotenti** ossia quelle che produccono lo stesso risultato anche se ripetute più volte.

In un server stateful si distingue lo stato in base alla durata massima:
- **stato permanente** mantenuto per sempre (autorizzazioni / username / password)
- **stato soft o a tempo** che rimane per un tempo massimo

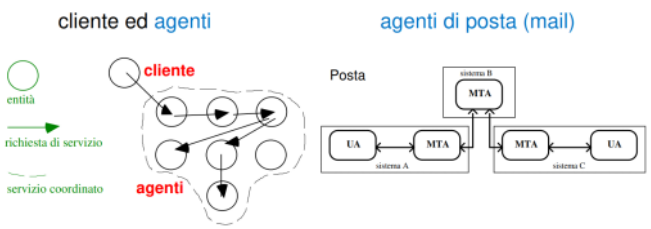
Concorrenza nel server

Una proprietà che caratterizza il server è la possibile concorrenza delle azioni, cioè la possibilità di portare avanti più operazioni soprattutto in caso di Web seerver che deve rispondere a molte richieste
Il server è tipicamente una sola attività e un solo processo, ma la concorrenza può migliorare le prestazioni. Si distinguono:

- **server iterativo** o sequenziale
- **server concorrente**
  - **monoprocesso** un unico processo server si divide tra il servizio della coda delle richieste e le operazioni
  - **multiprocesso** un processo server si occupa della coda delle richieste e genera processi figli, uno per ogni servizio

Modello ad agenti multipli

Gli agenti forniscono il servizio di coordinato e possono:
- partizionare le capacità di servizio
- replicare le funzionalità di servizio



Sistemi di nomi

sabato 7 dicembre 2019 13.33

I sistemi di nomi compensano la necessità di conoscenza reciproca delle entità: il cliente che vuole richiedere un servizio deve poter riferire il servitore.
I nomi possono essere trasparenti alla allocazione o non esserlo per consentire un più ato livello di astrazione

Un sistema di nomi gestisce ed accentra la funzione di **binding** cioè realizzare il legame tra la risorsa logica (nome) e la risorsa fisica (target) di riferimento.

- binding statico: i riferimenti risolti prima della esecuzione
- binding dinamico: i riferimenti risolti solo al momento del bisogno e durante l'esecuzione

Socket (linguaggio Java)

sabato 7 dicembre 2019 18.36

in Java è possibile programmare la rete attraverso meccanismi di visibilità della comunicazione (sul sistema operativo) contenuti in classi specifiche del package di networking **java.net**

Tipi di socket

- **datagram**: scambio di messaggi senza garanzie (best-effort o may-be) [✓ JVM]
- **stream**: scambio bidirezionale di messaggi in ordine, senza errori, non duplicati, nessun confine di messaggio, out-of-band flusso [✓ JVM]

Un servizio su un nodo è identificato da un nome globale composto da:
• indirizzo IP
• porta [0, 65535] (unsigned int a 16 bit)

Socket datagram (linguaggio Java)

sabato 7 dicembre 2019 18.44

DatagramSocket

Le socket datagram permettono a due thread di scambiarsi messaggi senza stabilire una connessione tra i thread coinvolti.
La classe DatagramSocket permette di definire tale socket e si usa senza distinzione sia per il client che per il server

```
DatagramSocket(InetAddress localaddress, int localPort) throws SocketException
```

Primitive di comunicazione

La send implica solo la consegna ad un livello di kernel locale che si occupa dell'invio (asincrona)

```
void send(DatagramPacket p);
```

La receive richiede una attesa del ricevente fino all'arrivo locale della informazione (sincrona)

```
void receive(DatagramPacket p);
```

DatagramPacket

La classe DatagramPacket permette di definire i datagrammi.

```
DatagramPacket(byte[] buf, int offset, int length, InetAddress address, int port)
```

InetAddress

La classe InetAddress rappresenta gli indirizzi logici e definisce una serie di metodi per ottenerli

Ottenere un oggetto InetAddress per l'host specificato:

```
static InetAddress getByName(String hostname);
```

Ottenere un array di oggetti InetAddress per più indirizzi IP sullo stesso nome logico:

```
static InetAddress[] getAllByName(String hostname);
```

Ottenere un oggetto InetAddress per macchina locale:

```
static InetAddress getLocalHost();
```

Esempio - codice per mittente

```
DatagramSocket socket = new DatagramSocket();
byte[] buf = {'C','i','a','o'};
InetAddress addr = InetAddress.getByName("137.204.59.72");
int port = 1900;
DatagramPacket packet = new DatagramPacket(buf, buf.length, addr, port);
socket.send(packet);
```

Esempio - codice per ricevente

```
InetAddress addr = InetAddress.getByName("137.204.59.72");
DatagramSocket socket = new DatagramSocket(addr, 1900);
int recport;
InetAddress recaddr;
byte[] res = new byte[200];
DatagramPacket packet = new DatagramPacket(res, res.length, recaddr, recport);
packet.setData(res);
socket.receive(packet);
recport = packet.getPort();
recaddress = packet.getAddress();
res = packet.getData();
```

La comunicazione a datagrammi consente anche l'uso di protocolli di gruppo, ossia protocolli di multicast.
Una comunicazione multicast permette di inviare messaggi a una serie di destinatari che sono registrati su un indirizzo di gr uppo (di ricezione).

MulticastSocket

Le funzionalità di multicast sono realizzate dalla classe MulticastSocket (basata su indirizzi IP di classe D: 224.0.0.0 - 239.255.255.255)

```
MulticastSocket(int port)
```

```
void joinGroup(InetAddress addr);
void leaveGroup(InetAddress addr);
```

Opzioni socket datagram

`setSoTimeout(int timeout)`	Definisce un timeout per l'esecuzione
`setSendBufferSize(int size)` `sendReceiverTimeoutSize(int size)`	Definisce le dimensioni del buffer mittente e/o ricevente
`setReuseAddress`	Si possono collegare più processi ad un certo indirizzo fisico

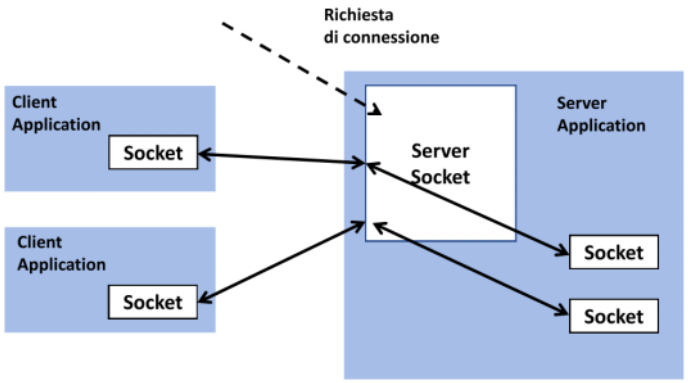
Socket a stream (linguaggio Java)

lunedì 14 ottobre 2019 18.35

Le socket stream sono i terminali di un canale di comunicazione virtuale, creato prima della comunicazione.
Hanno semantica **at-most once** ossia la ricezione avviene una volta sola anche se rinvio.
La comunicazione avviene in modo bidirezionale (full duplex), affidabile, con dati (byte) consegnati in sequenza.

La connessione tra i processi Client e Server è definita dal protocollo (in questo caso TCP-IP) e una quadrupla univoca:
indirizzo IP client; porta client; indirizzo IP server; porta server

Java mette a disposizione due classi distinte per la realizzazione della tecnologia:
`java.net.Socket`
`java.net.ServerSocket`



Socket

La classe Socket è usata per il cliente e consente di creare una socket "attiva" connessa a stream per il collegamento di un client ad un server.
Nota che la creazione della socket produce in modo atomico anche la connessione al server corrispondente che però deve essere presente.

```
Socket(String remoteHost, int remotePort) throws IOException
```

```
Socket(InetAddress remoteHost, int remotePort) throws IOException
```

```
Socket(InetAddress remoteHost, int remotePort, InetAddress localHost, int localPort) throws IOException
```

In caso di chiusura della socket la parte in IN viene eliminata subito mentre la parte OUT viene mantenuta per un certo tempo per continuare a spedire le informazioni al pari. Il pari si accorge che la parte in IN del corrispondente è stata chiusa mediante un eccezione in lettura/scrittura.
Chiudere la socket:

`void close()`	Chiude entrambi i canali di cominicazione dello stream
`void shutdownInput()`	Chiude il canale di input dello stream
`void shutdownOutput()`	Chiude il canale di output dello stream

Ottenere lo stream di lettura della socket:

```
InputStream getInputStream();
```

Ottenere lo stream di scrittura della socket:

```
OutputStream getOutputStream();
```

ServerSocket

La classe Serversocket è usata per il servitore e definisce una socket capace solo di accettare richieste di connessione provenienti da diversi Client

```
ServerSocket(int localPort) throws IOException
```

creare una socket in ascolto sulla porta specificata con una coda di lunghezza count:

```
ServerSocket(int localPort, int count) throws IOException
```

Ottenere la socket su cui avverrà la comunicazione tra client e server, la funzione è bloccante finchè non arriva una richiesta di connessione

```
Socket accept() throws IOException;
```

Funzioni di supporto

`InetAddress getInetAddress()`	Restituisce l'indirizzo del nodo remoto a cui la socket è connessa
`InetAddress getLocalAddress()`	Restituisce l'indirizzo della macchina locale
`int getPort()`	Restituisce il numero di porta sul nodo remoto cui la socket è connessa
`int getLocalPort()`	Restituisce il numero di porta localea cui la socket è legata

Socket a stream (linguaggio Java)

venerdì 18 ottobre 2019 13.10

DataOutputStream / DataInputStream

Queste classi mettono a disposizione una serie di metodi per l'invio e la ricezione di tipi primitivi Java:

DataOutputStream	DataInputStream
`void writeUTF(String str)`	`String readUTF()`
`void writeChar(char v)`	`char readChar()`
`void writeInt(int v)`	`int readInt()`
`void writeFloat(float v)`	`float readFloat()`

Stringhe UTF-16

Le stringhe in Java sono tipicamente accessibili attraverso UTF (Unified o Unicode Transformation Format) che permette il massimo dello standard. Tipicamente il supporto JVM memorizza le stringhe attraverso un formato interno UTF-16 (a 16 bit x carattere) e usa:

- un contatore dei caratteri della stringa
- un array di coppie di byte (due byte per ogni carattere della stringa)

Esempio - codice client

Per ogni ciclo si legge dallo standard input, si scrive sullo standard output e si attende sulla socket la risposta in echo:

```
try {
	oggSocket = new Socket(hostname, port);
	out = new PrintWriter(oggSocket.getOutputStream(), true);
	in = new BufferedReader(new InputStreamReader(oggSocket.getIntputStream()));
	userInput = new BufferedReader (new InputStreamReader(System.in));

	while((oggLine = userInput.readLine()) != null) {
		out.println(oggLine);
		System.out.println(in.readLine());
	}
	oggSocket.close();
}
catch (IOException e) {
	System.err.println(e);
}
```

Esempio - codice server

Ad ogni cliente il server sequenziale manda la data e chiude tutto:

```
try {
	oggServer = new ServerSocket(port);
	while (true) {
		oggConnessione = oggServer.accept();
		out = new PrintWriter (oggConnessione.getOutputStream(), true);
		Date now = new Date();
		out.write(now.toString()+ "\r\n");
		oggConnessione.close();
	}
}
catch (IOException e) {
	oggConnessione.close();
	oggServer.close();
	System.err.println(e);
}
```

Nel caso di server parallelo si genera un processo per ogni connessione accettata. Il nuovo processo ha la propria connessione ServerSocket

Opzioni socket stream

`setSoLinger(boolean on, int linger)`	Dopo la close, il sistema tenta di consegnarei pacchetti ancora in attesa di spedizione. Questa opzione consente di scartare i pacchetti in attesa dopo un intervallo di tempo linger in secondi
`setTcpNoDelay(boolean on)`	Il pacchetto viene inviato immediatamente senza bufferizzare
`setKeepAlive(boolean on)`	Attiva / disattiva l'opzione di keepalive

Socket (linguaggio C)

venerdì 18 ottobre 2019 14.32

UNIX deve fornire funzioni primitive di comunicazione (API sincrone e bloccanti)
UNIX Berkeley introduce il meccanismo di socket standard, come strumenti di comunicazione locale o remota con politiche differenziate, in alternativa ai problemi degli strumenti concentrati, trasparente e ben integrata con processi e file.

I processi possono scrivere/leggere messaggi e stream su socket, con molte opzioni e requisiti:

- **eterogeneità**: comunicazione fra processi su architetture diverse
- **trasparenza**: la comunicazione fra processi indipendentemente dalla localizzazione fisica
- **efficienza**: l'applicabilità delle socket limitata dalla sola performance
- **compatibilità**: i naive process (filtri) devono potere lavorare in ambienti distribuiti senza subire alcuna modifica
- **completezza**: protocolli di comunicazione diversi e differenziati

Per poter utilizzare le socket viene definito un nuovo tipo **Socket**, a cui è associato un intero univoco detto **Socket descriptor**; questo ci fa capire che abbiamo un limite alle socket che possiamo aprire su una macchina dettato dalla grandezza della tabella dei file aperti.
L'interfaccia per le socket è ispirata a quella dei file, infatti possiamo utilizzare la maggior parte delle funzioni associate ad essi:

`open()`	`close()`	`read()`	`write()`	`fctl()`	`loctl()`

Tipo di servizio

- **datagram**: scambio di messaggi senza garanzie (best-effort o may-be) [✓ UNIX]
- **stream**: scambio bidirezionale di messaggi in ordine, senza errori, non duplicati, nessun confine di messaggio, out-of-band flusso [✓ UNIX]
- **seqpacket**: messaggi con numero di ordine [✗ UNIX]
- **raw**: messaggi scambiati senza azioni aggiuntiva (per debug protocolli) [✗ UNIX]

Sistema di nomi

In UNIX c'è una differenziazione tra *nomi logici* delle socket (nomi locali) e *nomi fisici* (nomi globali). Una socket deve essere collegata al sistema fisico e richiedere un **binding**, cioè il legame tra socket logica ed entità fisica corrispondente.

{famiglia di indirizzo, indirizzo internet, numero di porta}

Tipi per indirizzi e nomi socket

```
struct sockaddr_in {
      u_short sin_family;
      u_short sin_port;
      struct in_addr sin_addr;
};

struct in_addr {
      u_long s_addr
};
```

Funzioni di supporto ai nomi

```
#include <netdb.h>
struct hostent * gethostbyname (char * name);
```

La funzione restituisce un puntatore ad una struttura hostent oppure NULL se fallisce, il parametro name viene ricercato nel file /etc/host.
La ricerca avviene localmente, poi integrata anche con strumenti (DNS).

```
struct hostent {
      char * h_name;                /* nome ufficiale dell'host */
      char ** h_aliases;            /* lista degli aliases */
      int h_addrtype;               /* tipo dell'indirizzo host */
      int h_length;                 /* lunghezza del'indirizzo */
      char ** h_addr_list;          /* lista indirizzi dai nomi host */
      #define h_addr h_addr_list[0] /* indirizzo nome host */
}
```

Esempio - utlizzo di gethostbyname

```
#include <netdb.h>

struct hostent *hp;
struct sockaddr_in peeraddr;
peeraddr.sin_family = AF_INET;
peeraddr.sin_port = 22375;

if(hp = gethostbyname(argv[1])) /* in caso di successo */
      peeraddr.sin_addr.s_addr = ((struct in_addr *)(hp->h_addr))->s_addr;
```

Socket (linguaggio C)

venerdì 18 ottobre 2019 15.43

```
struct servent {
      char * s_name;                /* nome ufficiale del servizio */
      char ** s_aliases;            /* lista degli aliases */
      int s_port;                   /* numero porta */
      int *s_proto;                 /* protocollo utilizzato */
}
```

```
#include <netdb.h>
```

```
struct servent * getservbyname(char * name, char * proto);
```

La funzione restituisce un puntatore alla struttura servent oppure NULL se fallisce, il parametro name viene ricercato nel file /etc/services

Esempio - utlizzo di gethostbyname

```
#include <netdb.h>
struct servent *sp;
struct sockaddr_in peeraddr;
peeraddr.sin_family = AF_INET;

sp = getservbyname("echo", "tcp");
peeraddr.sin_port = sp->s_port;
```

Primitive preliminari

```
int socket(int domain, int type, int protocol);
```

Restituisce un intero rappresentativo del file descriptor associato alla socket prendendo in ingresso il dominio (UNIX, Internet, etc.), il tipo (datagram, stream, etc.) e il protocollo

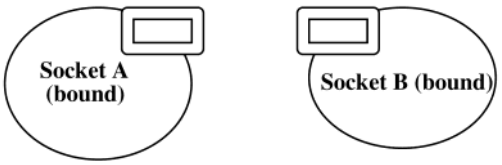
```
int bind(int sockfd, const struct sockaddr *addr, socklen_t addrlen);
```

Effettua il bind della socket e restituisce un valore positivo se andato a buon fine prendendo in ingresso il file descriptor associato alla socket, l'indirizzo locale della socket e la sua lunghezza

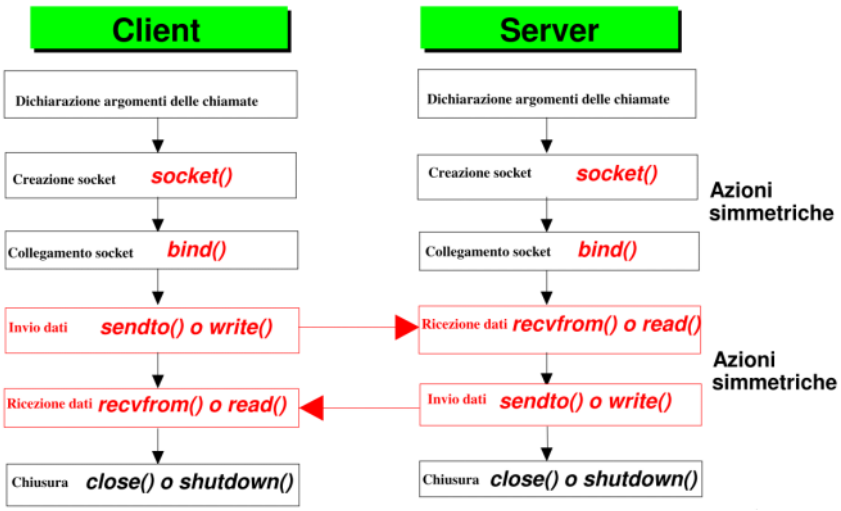
Socket datagram (lunguaggio C)

venerdì 18 ottobre 2019 16.04

Le socket datagram sono dei veri end-point di comunicazione e permettono di formare half-association (relative ad un solo processo), ma usabili per comunicare con chiunque del dominio.



Le socket datagram sono usate con un protocollo che si basa sulla sequenza di primitive seguente:



Primitive di comunicazione

```
int sendto(int s, char *msg, int len, int flags, struct sockadd_in *to, int *tolen);
```

Restituisce il numero di byte che sono stati inviati

```
int recvfrom(int s, char *buf, int len, int flags, struct sockadd_in *from, int *fromlen);
```

Restituisce il numero di byte che sono stati ricevuti

Esempio - server datagram:

```
int sockfd, n, len;
char msg[MAXLINE];
struct sockaddr_in server_address, client_address;
struct hostent * clienthost;
bzero(&server_address, sizeof(server_address);
server_address.sin_family = AF_INET;
server_address.sin_addr.s_addr = INADDR_ANY; // INADDR_ANY rappresenta tutte le interfacce disponibili
server_address.sin_port = 7;
sockfd = socket(AF_INET, SOCK_DGRAM, 0);
bind(sockfd, (struct sockaddr *) &server_address, sizeof(server_address))
while(true) {
      len = sizeof(struct sockaddr_in);
      n = recvfrom(sockfd, msg, MAXLINE, 0, &client_address, &len);
      clienthost = gethostbyaddr((char *) &client_address.sin_addr, sizeof(client_address.sin_addr), AF_INET);
      if (clienthost == NULL) printf("client host not found\n");
      m = sendto(sockfd, msg, n, 0, &client_address, len);
}
```

Funzioni accessorie socket

Gli interi sono composti da più byte e possono essere rappresentati in memoria secondo due modalità diverse di ordinamento:
Little-endian e *Big-endian*.
Per la rete il Network Byte Order (NBO) è standard ed è definito Big-endian, per gli host l' Host Byte Order (HBO) non è definito univocamente pertanto per mantenere la portbilità vengono utilizzate funzioni per la conversione di valori da rete a host (s: short , l: long)

- `shortlocale = ntohs(shortrete)`
- `longlocale = ntohl(longrete)`

e viceversa

- `shortrete = htons(shortlocale)`
- `longrete = htons(longlocale)`

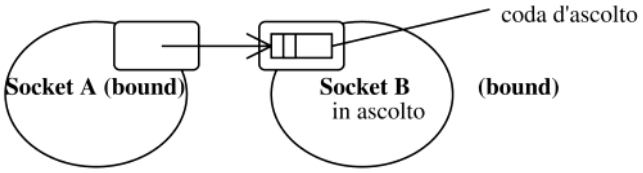
Per convertire da notazione con stringa a indirizzi IP a 32 bit e viceversa:

- `indirizzo = inet_addr(stringa)`
- `stringa = inet_ntoa(indirizzo)`

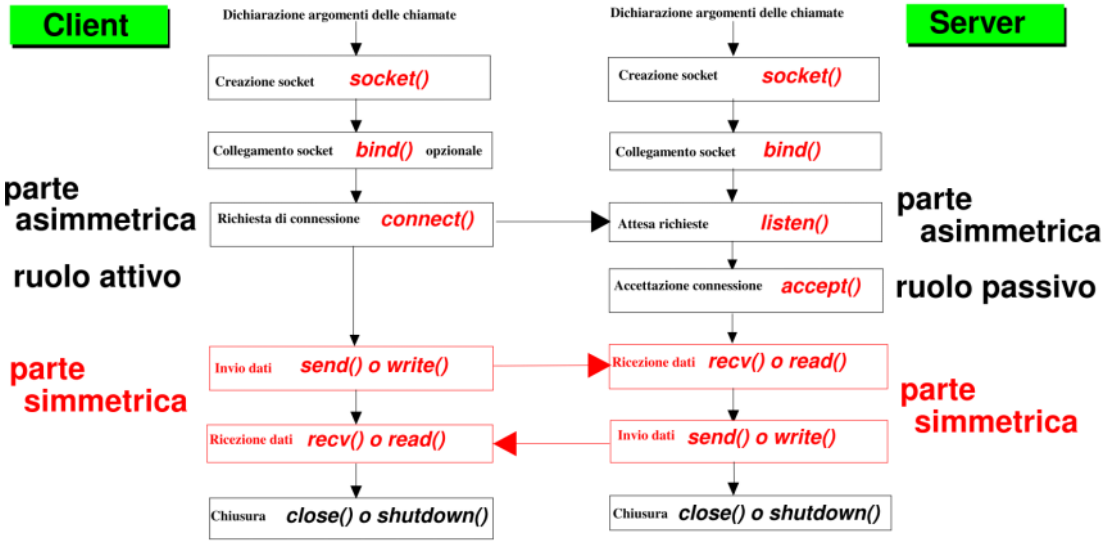
Socket stream (linguaggio C)

venerdì 18 ottobre 2019 17.37

Le socket stream prevedono una risorsa che rappresenta la connessione virtuale tra le entità interagenti



Le socket stream sono usate con un protocollo a sequenza differenziata di primitive e con ruoli diversi, per poi arrivare alla omogeneità dopo avere stabilito la connessione:



La connessione, una volta stabilita, permane fino alla chiusura di una delle due half-association, ossia alla decisione di uno due entità interagenti.

```
#include <sys/types.h>
#include <netinet/in.h>
#include <sys/socket.h>

int s, res;
int addrlen = sizeof(struc sockaddr_in);
struct sockaddr_in *addr;

s = socket(AF_INET, SOCK_STREAM, 0); /* fa solo la creazione locale */
res = bind(s, addr, addrlen);        /* collega la socket creata localmente alla porta e nodo globali */
```

Il client deve creare una connessione prima di comunicare

```
int connect(int s, struct sockaddr_in addr, int addrlen);
```

Restituisce un valore >= 0 nel caso sia stato possibile creare una connessione, altrimenti per valori negativi:

- ECOMM - communication error on send
- ECONNABORTED - connection aborted (non ritentare)
- ECONNREFUSED - impossibilità di connettersi (non ritentare)
- ETIMEDOUT - tentativo di connessione in time-out: la coda d'ascolto del server è piena o non creata. Non si è depositata la richiesta.

La primitiva connect() è anche capace di invocare la bind() e il sistema assegna al cliente la prima porta libera.

Il server deve creare una coda per possibili richieste di servizio

```
int listen(int s, int backlog);
```

Restituisce un valore positivo rappresentativo del file descriptor in caso di successo, altrimenti errore. I parametri sono rispettivamente il file descriptor della socket e la lunghezza della coda da creare: se un client tenta di connettersi e la coda è piena, quest'ultimo riceve un messaggio di errore.

Il server deve trattare ogni singola richiesta accodata

```
int accept(int s, struct sockaddr_in * addr, int * addrlen);
```

Restituisce un valore positivo rappresentativo di una nuova socket connessa al cliente, altrimenti errore. Si ottengono le informazioni relative al cliente attraverso i parametri addr e addrlen.

Socket stream (linguaggio C)

lunedì 21 ottobre 2019 14.15

Comunicazione sulla connessione

```
#include <sys/types.h>
#include <sys/socket.h>
```

```
int send(int s, char* msg, int length, int flags);
```

Restituisce il numero di byte inviati con successo in caso di valore positivo, altrimenti errore. I parametri presi in ingresso sono il descrittore della socket, il puntatore all'area del messaggio, la lunghezza del messaggio e dei flags

```
int recv(int s, char* buf, int length, int flags);
```

Restituisce il numero di byte ricevuti con successo in caso di valore positivo, altrimenti errore. I parametri presi in ingresso sono il descrittore della socket, il puntatore all'area del buffer, la lunghezza del messaggio e dei flags

flags:

- 0 normale
- MSG_OOB messaggio out-of-bound
- MSG_PEEK lettura non distruttiva dello stream

Analogamente possono essere utilizzate le funzioni:

```
int write(int s, char* msg, int length);
```

```
int read(int s, char* buf, int length);
```

I messaggi non sono comunicati ad ogni primitiva: i dati sono bufferizzati dal protocollo TCP: non è detto che siano inviati subito ma potrebbero essere raggruppati e inviati poi alla prima comunicazione 'vera' decisa dalla driver TCP.
L'unico modo per avere tale corrispondenza è inviare messaggi di lunghezza pari al buffer.

I messaggi inviati sono tipicamente a lunghezza fissa, per i messaggi a lunghezza variabile si alterna uno a lunghezza fissa contenente la lunghezza del secondo, poi il messaggio a lunghezza variabile.

Si coopera alla chiusura della connessione usando dei segnali di fine flusso:
Su ogni flusso viaggiano dati in stream fino alla fine del file del pari di autorità che, dopo avere operato completamente segnala con una chiusura che non ci si devono aspettare più dati (via fine file) e si aspetta che i dati siano consumati.

Per non impegnare risorse non necessarie, esse vanno rilasciate con la primitiva:

```
int close(int s);
```

La primitiva è istantanea per il processo invocante ma la driver TCP può rimanere allocata ancora per del tempo dopo la chiusura della socket.
La close elimina tutto il contenuto del buffer in input ma lascia invariato il buffer in output per far si che eventuali dati possano essere ancora inviati.
La memoria del buffer di trasporto (porta) persiste e può rimanere piena anche alla terminazione del processo nel caso di impossibilità di invio.
Nota: Per questo motivo è importante fare la reuse-address quando il server va in crash: per potersi ricollegare su quella porta e riprendere l'invio.

Dopo la close (e la notifica), il pari connesso alla socket chiusa:

- se legge dalla socket, ottiene finefile
- se scrive ottiene un segnale di connessione non più esistente
- la durata della close per la driver TCP è poco controllata e può impegnare le risorse anche per molto tempo (minuti)

Un'alternativa alla close è la primitiva shutdown() che permette di terminare la connessione con una migliore gestione delle risorse sui due versi

```
int shutdown(int s, int how);
```

- how = 0, SHUT_RD -> Non si ricevono più dati, ma si può trasmettere. La send() del pari ritorna con -1 ed il processo riceve SIGPIPE.
- how = 1, SHUT_WR -> Si può solo ricevere dati dalla socket e non trasmettere. L'altro pari collegato alla connessione alla lettura riceve EOF
- how = 2, SHUT_RDWR -> Entrambi gli effetti

Opzioni per le socket

```
int getsockopt(int s, int level, int optname, int * optval, int * optlen);
int setsockopt(int s, int level, int optname, int * optval, int optlen);
```

Opzioni	Descrizione	Opzioni	Descrizione
SO_REUSEADDR	Riuso dell'indirizzo locale	SO_SNDTIMEO	Setta il timeout dell'output
SO_SNDBUF	Setta dimensione del buffer in output	SO_RCVTIMEO	Setta il timeout dell'input
SO_RCVBUF	Setta dimensione del buffer in input		

SO_KEEPALIVE	Si chiede di tener viva la connessione anche se non viene utilizzata per lunghi periodi di tempo
SO_LINGER	Si setta un tempo di vita al buffer della socket per liberare risorse dopo la close()

Socket stream (linguaggio C)

lunedì 28 ottobre 2019 17.39

Macro condizionali

Sono istruzioni per il compilatore e ci servono poiché le funzioni che andremo successivamente ad usare sono fortemenți dipendenti dal kernel:

```
#ifdef __linux__
     ...
#endif
```

Modalità primitive socket

E' spesso importante poter usare modalità non sincrone bloccanti che non prevedono attesa del completamento (ad esempio effettuare altre operazioni durante una lettura in input).
Le socket asincrone permettono operazioni senza attesa, ma al completamento tipicamente l'utente viene avvisato con un segnale **SIGIO** che segnala un cambiamento di stato della socket per l'arrivo di dati (e viene ignorato dai processi che non hanno definito un gestore).

```
#include <sys/ioctl.h>
int ioctl(int filedesc, int request, ... /* args */)
```

Per poter definire a chi deve essere consegnato il segnale SIGIO in un gruppo di processi si utilizza l'attributo **SIOCSPGRP**:
`int ioctl (filedescr, SIOCSPGRP, &flag)`
flag < 0: il segnale solo ad un processo con pid uguale al valore negato
flag > 0: il segnale arriva a tutti i processi del process group

Esempio di assegnazione dell'attributo non bloccante:

```
#include <sys/ioctl.h>
int s;       /* socket descriptor */
int arg = 1; /* valore per la qualifica non blocking */
ioctl(s, FIONBIO, &arg);
ioctl(s, FIOASYNC, &arg);
```

Se rendiamo una socket asincrona o sincrona non bloccante, ci troviamo in una situazione di uso in cui le operazioni vengono richieste ma il processo non si sospende in attesa della terminazione.
Nel caso di azioni di lettura ci serve a poco, poiché siamo interessati al dato che arriva da altri.
Per il caso di azioni di scrittura noi possiamo comandarle e non aspettare il termine

Intercettazione di eventi

La select() invocata sospende il processo fino al primo evento o al timeout (se specificato) o attende il primo evento (sincrona con il primo):

```
#include <time.h>
int select(size_t nfds, int * readfds, int * writefds, int * exceptfds, const struct timeval * timeout);
```

Il parametro nfds è il numero max di eventi attesi, timeout è impostato a null se si desidera attesa indefinita (comportamento di default).
Gli altri tre parametri sono degli array di interi in cui ogni posizione rappresenta un file descriptor:

9	*8*	*7*	*6*	*5*	*4*	*3*	*2*	*1*	*0*	*posizione file descriptor*
1	0	**1**	0	**1**	**1**	0	0	0	0	maschera ingresso
0	0	**1**	0	**1**	**0**	0	0	0	0	maschera uscita

Restituisce il numero di eventi che si sono verificati.

Funzioni di supporto

```
void FD_SET(int fd, fd_set &fd_set);
```

Include la posizione fd di fd_set e la setta a valore 1

```
void FD_CLR(int fd, fd_set &fd_set);
```

Rimuove la posizione fd di fd_set

```
int FD_ISSET(int fd, fd_set &fd_set);
```

Verifica se la posizione fd di fd_set è settata a 1 o 0

```
void FD_ZERO(fd_set &fd_set);
```

Inizializza l'insieme dei descrittori a zero

Server multifunzione

Spesso è significativo avere un unico servitore per più servizi come un unico collettore attivo che si incarica di smistare le richieste. Il problema potrebbe sussistere nel caso in cui si incarichi il server di azioni molto onerose, in tal caso il server potrebbe diventare il collo di bottiglia del sistema.
Una possibile soluzione potrebbe essere far eseguire al server multifunzione le azioni più semplici (internal) mentre quelle più complesse vengono delegate a dei processi figli.

Alcuni servizi applicativi

lunedì 4 novembre 2019 16.15

telnet

E' un servizio di terminale remoto.
Cliente:
1. Stabilisce una connessione TCP con Server, accetta i caratteri dall'utente e li invia al Server
2. Accetta i caratteri del server in risposta e li visualizza sul terminale d'utente

Server:
1. Creazione demone sul server per la gestione del servizio (accettare richieste di connessione)
2. Generazione di un processo figlio (da parte del demone) per la gestione delle singole sessioni (ricezione dati, invio al sistema locale, invio risposta al client)

Sia telnet, sia ftp sono basati su un modello di terminale detto standard **Network Virtual Terminal** (NVT) per risolvere il problema di eterogeneità dei terminali: client e server devono tradurre il formato dei dati in tale standard in invio e tradurre nuovamente dallo standard NVT a quello di sistema in ricezione *(livello di presentazione)*.

rlogin

E' un servizio di login remoto.

ftp

E' un servizio per il trasferimento di file e accesso (File Transfer Protocol).
Si affida al protocollo TCP perché più affidabile ma esiste anche la variante tftp (Trivial File Transfer Protocol) basato su UDP

mail

Il servizio di mail si basa sul protocollo RFC822 che è di tipo legacy (non modificabile) e ammette solo l'invio di caratteri ASCII. E' stato esteso con il protocollo Multiporpose Interchange Mail Extension (MIME) che permette l'invio di messaggi con formati diversi.
L'utente attraverso un servizio applicativo si collega all' User Agent (UA). Tra user agent e Mail transport Agent (MTA) insiste una connessione, come pure tra MTA e MTA.

Simple Mail Transfer Protocol RFC821 (SMTP)

È il protocollo standard per il trasferimento della mail tra mailer (MTA) che si connettono e scambiano messaggi di posta in chiaro

news

News come un insieme di gruppi di discussione collettiva. Ogni gruppo riguarda un particolare argomento.
Alcuni dei principali gruppi di news:

comp	computer	misc	miscellaneous
news	news	biz	business
rec	recreative	soc	society
sci	science	talk	talk
alt	alternative	bit	bitnet

Network News Transfer Protocol (NNTP)

È il protocollo standard per il trasferimento delle news tra agenti

Standard di comunicazione OSI

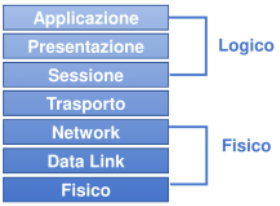
lunedì 4 novembre 2019 17.25

OSI (Opem System Interconnections) nasce con obiettivi di razionalizzazione per qualunque tipo di comunicazione tra sistemi diversi.
OSI è uno standard per la gestione dei sistemi remoti (System management o Network management)

OSI propone standard e schemi di progetto astratti, per razionalizzare, inquadrare e abilitare ogni comunicazione.
OSI è:

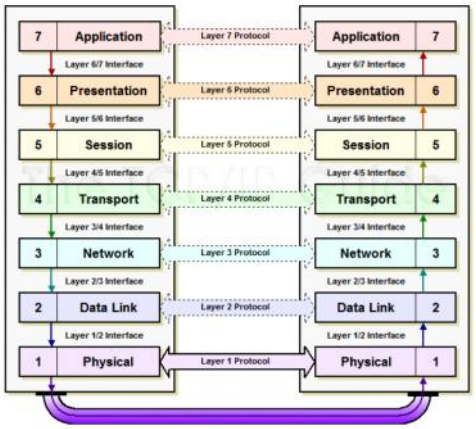
- organizzato a livelli, con precisi obiettivi e significati
- interamente ad oggetti
- astratto e senza legami con realizzazioni (proprietarie o meno)

L'architettura si basa sul principio dell'astrazione che richiede di mostrare solo le entità utili per l'utente finale (cliente).
L'astrazione prevede una struttura suddivisa in livelli che permettono di affrontare le complessità separatamente.



Ogni livello inferiore nasconde al livello superiore dei dettagli.
Ogni livello superiore usa gli strumenti esposti da quello inferiore per il suo obiettivo.
Ogni livello prevede:

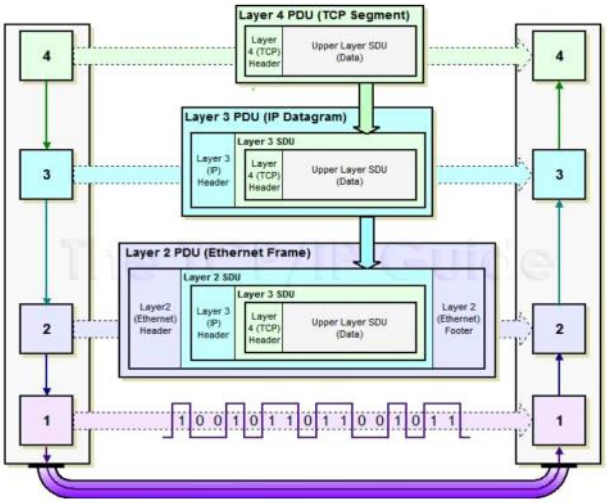
- un protocollo da realizzare definendo lo standard di protocollo di quel livello (visione orizzontale)
- un servizio da presentare/richiedere tramite interfaccia (visione verticale)



Il **Service Access Point** (SAP) identifica il punto di accesso che un servizio OSI offre al suo livello superiore.
Ogni SAP deve avere un nome identificativo unico. Possono esserci anche più SAP per uno stesso livello.

OSI definisce solo le specifiche di comunicazione senza suggerire alcuna tecnologia di realizzazione.
Infatti mentre i protocolli e le architetture reali cambiano in modo anche veloce, è stato necessario definire uno standard che rimanesse a lungo.
Vengono standardizzati i formati dei messaggi scambiati tra i diversi livelli. I tipi di data unit standardizzati sono:

- Service Data Unit (SDU) della sessione per richiedere servizi
- Interface Data Unit (IDU) come richieste portate all'interfaccia
- Protocol Data Unit (PDU) con cui si realizzaono operazioni con il pari orizzontale



PDU fornito dal livello diventa SDU per il livello sottostante.

Standard di comunicazione OSI

venerdì 22 novembre 2019 11.28

Connessione

Un sistema può essere:

- connectionless
  Ogni unità di dati è trasferita in modo indipendente dalle altre (senza ordine). Non viene garantita nessuna qualità del servizio e nessuna negoziazione sullo scambio di informazioni.
- connection-oriented
  Si stabilisce una connessione tra entità pari che devono comunicare.

Livelli e compiti

#	Layer Name	Key Responsibilities	Data Type Handled	Scope	Common Protocols and Technologies
1	Physical	Encoding and Signaling; Physical Data Transmission; Hardware Specifications; Topology and Design	Bits	Electrical or light signals sent between local devices	(Physical layers of most of the technologies listed for the data link layer)
2	Data Link	Logical Link Control; Media Access Control; Data Framing; Addressing; Error Detection and Handling; Defining Requirements of Physical Layer	Frames	Low-level data messages between local devices	IEEE 802.2 LLC, Ethernet Family; Token Ring; FDDI and CDDI; IEEE 802.11 (WLAN, Wi-Fi); HomePNA; HomeRF; ATM; SLIP and PPP
3	Network	Logical Addressing; Routing; Datagram Encapsulation; Fragmentation and Reassembly; Error Handling and Diagnostics	Datagrams / Packets	Messages between local or remote devices	IP; IPv6; IP NAT; IPsec; Mobile IP; ICMP; IPX; DLC; PLP; Routing protocols such as RIP and BGP
4	Transport	Process-Level Addressing; Multiplexing/Demultiplexing; Connections; Segmentation and Reassembly; Acknowledgments and Retransmissions;Flow Control	Datagrams / Segments	Communication between software processes	TCP and UDP; SPX; NetBEUI/NBF
5	Session	Session Establishment, Management and Termination	Sessions	Sessions between local or remote devices	NetBIOS, Sockets, Named Pipes, RPC
6	Presentation	Data Translation; Compression and Encryption	Encoded User Data	Application data representations	SSL; Shells and Redirectors; MIME
7	Application	User Application Services	User Data	Application data	DNS; NFS; BOOTP; DHCP; SNMP; RMON; FTP; TFTP; SMTP; POP3; IMAP; NNTP; HTTP; Telnet

Livelli logici

Sessione

La sessione determina i meccanismi per il dialogo tra entità diverse.
Un dialogo può avere molte dimensioni possibili (scambio testo tipo chat, scambio file, video in real-time, …).
Strutturazione del dialogo attraverso oggetti astratti detti **token**, intesi come gettoni di autorizzazione: un solo utente possiede il token in ogni momento ed ha il diritto di uso di un insieme di servizi di Sessione.

Presentazione

La presentazione affronta il problema della rappresentazione dei dati, e delle differenze tra sistemi che comunicano (ambienti e linguaggi di programmazione diversi, SO diversi, architetture diverse), ma anche dei casi di necessità di codifiche ad hoc per compressione dei dati (efficienza) o crittografia (sicurezza).
Tale funzionalità può essere realizzata mediante due soluzioni:

- dotare ogni nodo di tutte le funzioni di conversione possibili per ogni possibile rappresentazione dei dati
- concordare un formato comune di rappresentazione dei dati (usata effettivamente)

Problema uniformità del formato dati

Nel caso di eterogeneità nel formato dei dati tra mittente e ricevente occorre trasformare il dato in un formato standard e poi tradurre lo standard nel formato locale dell'architettura (Java lo fa in automatico).

Nel caso ci sia già un accordo basta pensare di udare diversi gradi di ridondanza a sencodo del costo associato alla comunicazione (impiego di banda). Ad esempio:

- lunghezza del valore, valore
- tipo, lunghezza del valore, valore

Nel caso non c'è stato un accordo in precedenza sul formato standard da utilizzare occorre accordarci sfruttando un apposito protocollo di descrizione dei dati.

Protocollo di negoziazione

Protocolli con numero di fasi non predicibile, spesso determinato da eventi verificatasi durante il protocollo stesso.
Basato su asta (bidding) tra sender e receiver. Fasi:

1. sender fa un broadcast della propria esigenza
2. i receiver fanno un'offerta (bid)
3. il sender sceglie
4. il receiver accoglie l'ok definitivo
5. accordo (award)

Nota: Protocollo di livello applicativo ma ultimamente viene integrato in sistemi anche di più basso livello come DHCP

Standard di comunicazione OSI

lunedì 11 novembre 2019 16.41

Non è detto che la strategia migliore venga adottata: il sender potrebbe accogliere un'offerta conveniente ma potrebbe arrivarne subito dopo una ancora migliore.

Il lvello di presentazione definisce:

- un linguaggio astratto di specifica: Abstract Syntax Notation 1 (ASN1) usato per sistemi dinamici
- un linguaggio concreto di descrizione: Basic Ecnoding Rules (BER) usato estensivament

BER

Triple Tag-Length-Value:

codifica a discesa ricorsiva

```
address.source = " Suna"
address.destination = "Decb"
length = 3
data = 'x', 'y', 'z'
```

Primitive types	BER
BOOLEAN	
INTEGER	02
OCTETSTRING	04
IA5String	16
Constructor types	
SEQUENCE	30
SEQUENCE OF	
SET	
SET OF	
CHOICE	

```
30 22
     30 12
          16 04 'S' 'u' 'n' 'a'
          16 04 'D' 'e' 'c' 'b'
     02 01 03
     04 03 01 02 03
```

Nota: BER e ANS1 non sono linguaggi general purpose ma linguaggi di dati.

ANS1

Primitive types
BOOLEAN
INTEGER
OCTETSTRING
IA5String

Constructor types
SEQUENCE
SEQUENCE OF
SET
SET OF
CHOICE

```
Address ::= SEQUENCE {
        addr_src    IA5String,
        addr_dst    IA5String
}

Pdu ::= SEQUENCE {
        pdu_ad      Address,
        pdu_len     INTEGER,
        pdu_data    OCTETSTRING (SIZE 1024)
}
```

Nota: E' in grado di scambiare anche codice molto semplice oltre ai dati

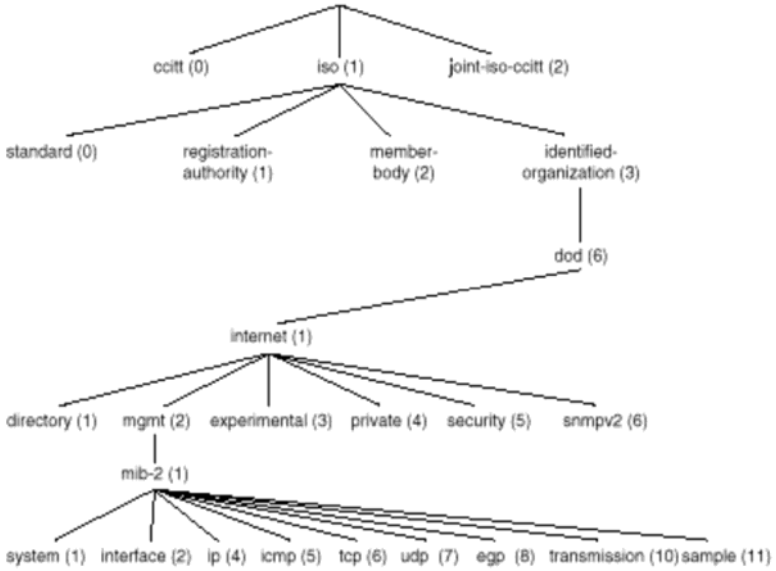
Applicazione

Il livello di Applicazione è il livello che si interfaccia con l'utente finale della comunicazione in base al modello OSI.
L'obiettivo è l'astrazione: nascondere la compressità dei livelli sottostanti.

OSI adotta un approccio particolare basato sul modello ad Oggetti per la specifica delle applicazioni.
E' necessaria l'unicità dei nomi dei servizi per l'accordo e il coordinamento

Servizio X.500

Il servizio di direttorio consente di collocare e classificare ogni entità di interesse (ogni dispositivo noto) in base al contenuto degli attributi in un sistema gerarchico di conoscenza molto accessibile (24/7).



Java Remote Method Invocation (RMI)

lunedì 11 novembre 2019 17.48

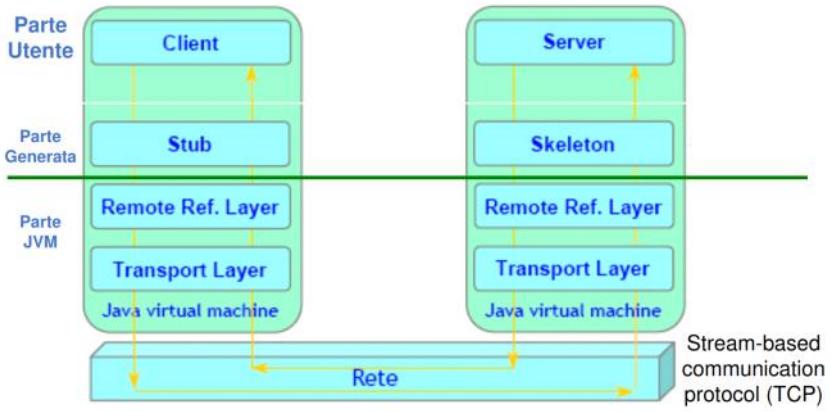
L'architettura RMI introduce la possibilità di richiedere esecuzione di metodi remoti in JAVA (RPC in JAVA) integrando il tutto con il paradigma OO.
Permette ad un'applicazione Java in esecuzione su una macchina di invocare i metodi di un oggetto di una applicazione Java in esecuzione su una macchina remota.

Viene creato localmente solo il riferimento ad un oggetto remoto, che è invece effettivamente attivo su un nodo remoto.
Un programma cliente invoca i metodi attraverso questo riferimento locale mantenuto in una variabile interfaccia.

Variabili di interfaccia

Una variabile tipata come tutte quelle classi che implementano quell'interfaccia

Non è possibile riferire direttamente l'oggetto remoto ma si necessita di una infrastruttura: il proxy.
Il proxy del cliente è detto **stub** e il proxy del servitore è detto **skeleton**.
Tutto quello che riguarda la comunicazione tra i due proxy viene generato automaticamente dalla JVM:



Remote Reference Layer (**RRL**):
Responsabile della gestione dei riferimenti agli oggetti remoti, dei parametri e delle astrazioni di stream-oriented connection. Ad esempio se devo passare un riferimento ad un oggetto locale su client ad un metodo del server se ne fa carico tale livello.

Utilizziamo una comunicazione basata su TCP per via della mole di dati che vanno scambiati in tal caso tra client e server.

Implementazione interfaccia

L'interfaccia deve estendere l'interfaccia Remote

```
public interface EchoInterface extends java.rmi.Remote
```

Ciascun metodo remoto deve lanciare una RemoteException

```
String getEcho(String echo) throws java.rmi.RemoteException
```

Implementazione Server

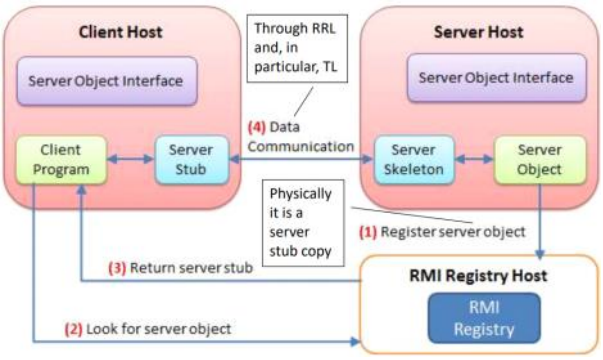
La classe che implementa il server deve estendere la classe UnicastRemoteObject e implementare tutti i metodi definiti nell'interfaccia implementata

```
public class EchoRMIServer extends java.rmi.server.UnicastRemoteObject implements EchoInterface

...

EchoRMIServer serverRMI = new EchoRMIServer();
Naming.rebind("localhost:1099/EchoService", serverRMI);
```

Nota: La porta 1099 è la porta di default del registry

Implementazione Client

```
EchoRMIServer serverRMI = (EchoInterface)Naming.lookup("localhost:1099/EchoService");
String echo = serverRMI.getEcho("Hello World!");
```



RMI Registry

giovedì 14 novembre 2019 10.10

Localizzazione del servizio: un client in esecuzione su una macchina ha bisogno di localizzare un server a cui connettersi, che è in esecuzione su un'altra macchina. L'RMI Registry gioca il ruolo di spazio di nomi (**naming service**) in una locazione ben nota, che il client conosce.

La java RMI utilizza un naming service detto RMI Registry (default port: 1099) che mantiene un insieme di coppie
`{name, reference}`

name è una stringa del tipo:
`//nomehost:porta/sevizio`
Nota: NON è il percorso, non c'è trasperenza alla locazione

Attivazione registry

1. occorre attivare il registry (di default sulla porta 1099 oppure specificandola direttamente)

```
rmiregistry 12345
```

Nota: Così facendo il registry è attivato in una nuova istanza JVM separata da quella del server

2. lato server, una volta compilati interfaccia e oggetto remoto, occorre generare gli eseguibili stub e skeleton

```
rmic -vcompat EchoRMIServer
```

3. si può avviare il server e i client

Procedura di comunicazione

Lo stub:
1. effettua la serializzazione delle informazioni per la chiamata (id del metodo, identificazione e argomenti)
2. invia le informazioni allo skeleton utilizzando le astrazioni messe a disposizione dal **Remote Reference Layer** (RRL)

Lo skeleton:
3. effettua la de-serializzazione dei dati ricevuti
4. invoca la chiamata sull'oggetto che implementa il server (dispatching)
5. effettua la serializzazione del valore di ritorno e lo invia allo stub

Lo stub:
6. effettua la de-serializzazione del valore di ritorno
7. restituisce il risultato al client

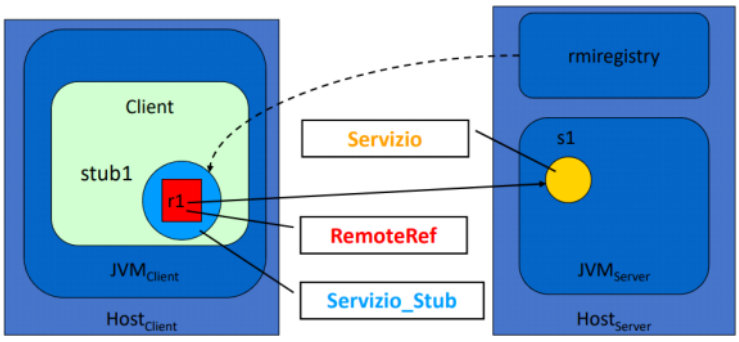
Serializzazione

Stub e skeleton utilizzano il bytecode Java uniforme e standard per serializzare e deserializzare (writeObject()/readObject()) gli oggetti di scambio tra client e server.

Passaggio di parametri

Tipo	Metodo locale	Metodo remoto
primitivo	per valore	per valore
oggetto	per riferimento	per valore (interfaccia Serializable deep copy)
oggetto remoto		per riferimento remoto (interfaccia Remote)

Quando passo un parametro per riferimento remoto sto idealmente passando il riferimento allo stub. Il codice non viene trasmesso quindi: la classe e i metodi rimangono assolutamente locali.

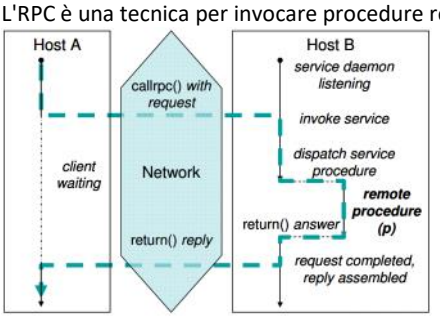


Il Client accede al Server RMI implementato dalla classe Servizio attraverso il riferimento remoto cioè lo stub1 (istanza della classe Servizio_Stub e passata dall'rmiregistry al client). Servizio_Stub contiene al suo interno un RemoteRef (r1) che consente al RRL di raggiungere il server.

Remote Process Calls (RPC) di SUN

lunedì 25 novembre 2019 16.26

L'RPC è una tecnica per invocare procedure remote



Le RPC sono basate su un contratto esplicito sulle informazioni scambiate, che consiste di due parti descrittive:
1. definizioni di programmi RPC: specifiche del protocollo RPC per i servizi offerti, cioè l'identificazione dei servizi ed il tipo dei parametri
2. definizioni XDR: definizioni dei tipi di dati dei parametri. Presenti solo se il tipo di dato non è un tipo noto in RPC
Entrambe raggruppate in un file con estensione .x cioè XDR sorgente
Esempio:

```
/* stampa.x */
program STAMPAPROG {
      version STAMPAVERS {
            int PRINTMESSAGE(string) = 1;
      } = 1;
} = 0x20000013;
```

Il protocollo XDR è dunque il mezzo utilizzato per mantenere l'omogeneità tra architetture differeti e svolge le funzioni di Marshalling e si colloca in OSI a livello di presentazione.

Funzioni built-in

Funzione built-in	Tipo di dato
xdr_bool()	Logico
xdr_char() *xdr_u_char()*	Carattere
xdr_short() *xdr_u_short()*	Intero a 16 bit
xdr_enum()	Enumerazione
xdr_float()	Virgola mobile
xdr_int(), xdr_u_int	Intero
xdr_long(), *xdr_u_long()*	Intero a 32 bit
xdr_void()	Nullo
xdr_opaque()	Opaco (raw byte)
xdr_double()	Doppia precisione

Funzioni per tipi composti

Funzione	Tipo di dato
xdr_array()	Vettori con elementi di tipo qualsiasi
xdr_vector()	Vettori a lunghezza fissa
xdr_string()	Sequenza di caratteri con terminatore a NULL ***N.B.: stringa in C***
xdr_bytes()	Vettore di bytes senza terminatore
xdr_reference()	Riferimento ad un dato
xdr_pointer()	Riferimento ad un dato, incluso NULL
xdr_union()	Unioni

In XDR non è possibile definire strutture innestate come matrici ma occorre passare per strutture intermedie definendo ad esempio una riga e poi un vettore di righe.

RPC di SUN prevede procedure con un unico argomento in ingresso e in uscita.
Il protocollo non prevede la concorrenza , il server è sequenziale e ad ogni invocazione il client è sospeso in attesa della risposta.

Step da svolgere:

1. Definire servizi e tipi di dati (se necessario) → esempio.x
2. Generare in modo automatico gli stub del client e del server e (se necessario) le funzioni di conversione XDR → **rpcgen** esempio.x
3. Realizzare i programmi client e server includendo il file esempio.h generato
4. Compilare tutti i file sorgente (client, server, stub e file per la conversione dei dati)
5. Pubblicare i servizi (lato server)
    a. attivare il **portmapper** (l'equivalente al registry RMI)
    b. registrare i servizi presso il portmapper (eseguendo il server).
6. Reperire (lato client) l'endpoint del server tramite il portmapper e creare il gestore di trasporto per l'interazione col server

Le procedure registrate sul server devono usare variabili di ritorno statiche in modo da essere disponibile e poter operare marshalling e spedizione anche quando la procedura termina. Se non fossero static esse verrrebbero rimosse dallo stack

RPC - client

giovedì 28 novembre 2019 09.47

- *livello intermedio*

Chiamata al meccanismo RPC: provoca l'esecuzione della procedura remota

```
int callrpc (char * remotehost, u_long n_prog, u_long n_vers, u_long n_proc, xdrproc_t xdr_arg, char * arg,
                                    xdrproc_t xdr_res, char * res)
```

remotehost	host remoto
n_prog n_vers n_proc	programma, versione e protocollo che identificano la procedura
xdr_arg xdr_res	tipo dell'argomento tipo del risultato
arg	argomento
res	risultato

Restituisce 0 in caso di successo o un intero indicante l'errore

- *livello basso*

1. Creazione del gestore di trasporto per il client

```
CLIENT * clnt_create(char * host, u_long n_prog, u_long n_vers, char * protocol)
```

host	nome del nodo remoto (server)
n_prog n_vers	programma e versione che identificano la procedura remota
protocol	protocollo da utilizzare ("tcp" o "udp")

2. Chiamata alla procedura remota

```
enum clnt_stat clnt_call (CLIENT * clnt, u_long n_proc, xdrproc_t inproc, char * in, xdrproc_t * outproc,
                                    char * out, struct timeval tout)
```

clnt	gestore di trasporto locale
n_proc	identificativo della procedura
inproc	tipo dell'argomento
outproc	tipo del risultato
in	argomento
out	risultato
tout	timeout di attesa della risposta

Definendo un timeout nullo il client può continuare immediatamente l'esecuzione (**modalità asincrona**).
Prerogative: il servitore non deve prevedere risposta (outproc deve essere xdr_void e out deve essere NULL) e il protocollo deve essere TCP .
Tale modalità detta asincrona **batch** si affida non sul livello applicativo ma sulla solidità del livello di trasporto: tutte le richieste vengono poste nel buffer TCP e gestite dalla driver di trasporto senza bloccare il processo che le genera.
L'ultima chiamata deve essere sincrona per essere sicuri di svuotare il buffer TCP (potrebbero rimanere pacchetti non inviati se non è stato riempito).

3. Stampa di eventuali errori

```
void clnt_perror(CLIENT* clnt, char * s)
```

clnt	riferimento al gestore di trasporto
s	stringa di output

4. Distruzione del gestore di trasporto per il client

```
void clnt_destroy(CLIENT * clnt);
```

clnt	riferimento al gestore da eliminare

RPC - server

giovedì 28 novembre 2019 20.39

- *livello intermedio*

Associa un identificatore unico alla procedura remota implementata nell'applicazione.

```
int registerrpc (u_long n_prog, u_long n_vers, u_long n_proc, char * (* proc_name() ), xdrproc_t xdr_arg,
                                                xdrproc_t xdr_res)
```

n_prog n_vers n_proc	programma, versione e protocollo che identificano la procedura
proc_name	identificatore alla procedura da attivare
xdr_arg	tipo dell'argomento
xdr_res	tipo del risultato

Ogni associazione si basa su una tripla [programma, versione, protocollo] a cui viene associata una porta. La struttura di registrazione detta **port map** del portmapper è una lista linkata per effettuare l'eliminazione di un servizio in maniera più efficiente.

- *livello basso*

1. Creazione del gestore di trasporto per il server

```
SVCXPRT * svcudp_create (int sock)
```

sock	socket o RPC_ANYSOCK per generarla in maniera automatica

```
SVCXPRT * svctcp_create (int sock, u_int send_buf_size, u_int recv_buf_size )
```

sock	socket o RPC_ANYSOCK per generarla in maniera automatica
send_buf_size recv_buf_size	dimensioni dei buffer di scambio dati

2. registazione del servizio sul portmapper

```
bool_t svc_register (SVCXPRT * xprt, u_long n_prog, u_long n_vers, char (* dispatch() ), u_long n_prot)
```

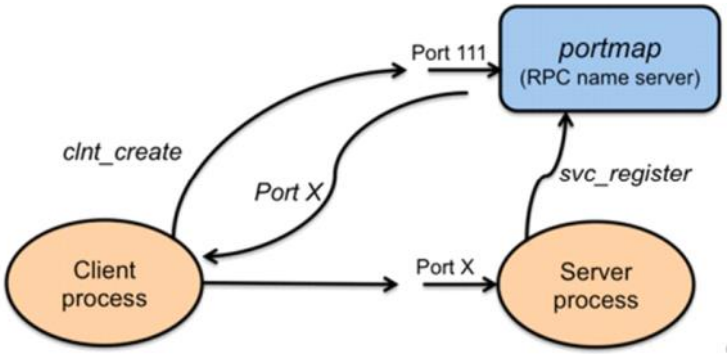
xptr	gestore di trasporto locale
n_prog n_vers n_prot	programma, versione e protocollo che identificano la procedura
dispatch	puntatore alla procedura di dispatching

3. Attesa infinita di chiamate (realizzata come una select in un ciclo infinito)

```
void svc_run()
```

Portmapper

Il portmapper è il server di nomi, lui stesso è un server RPC ed è reperibile alla porta 111
Lista di tutti i servizi invocabili su di un host remoto:
rpcinfo -p nomehost



Internet e protocolli

venerdì 6 dicembre 2019 14.00

TCP - *Transmission Control Protocol* - livello TX
flusso di byte bidirezionale a canale virtuale best effort, dati non duplicati, affidabili, con controllo di flusso

UDP - *User Datagram Protocol* - livello TX
scambio di messaggi end-2-end. I datagrammi possono essere persi, duplicati, pesantemente ritardati o consegnati fuori ordine

IP - *Internet Protocol (Routing)* - livello di RETE
scambio di datagrammi senza garanzia di consegna tra vicini

ICMP - *Internet Control Message Protocol* - gestione RETE
scambio di messaggi di controllo per errori IP. Errori su messaggi ICMP non causano ovviamente altri messaggi ICMP

ARP e RARP Protocol - (Reverse) *Address Resolution Protocol* - livello fisico
associazione di nomi fisici (MAC address) e nomi logici (IP address).
ARP: da IP trovare MAC
RARP : da MAC trovare IP (l'indirizzo IP viene mantenuto su disco; le macchine diskless lo richiedevano a un server di rete)
RARP è ormai deprecato

DHCP - *Dynamic Host Configuration Protocol*
attribuzione dinamica di indirizzi IP. Protocollo di offerta modello asta (bidding) a più fasi con iniziativa del cliente:

1. broadcast della richiesta di (discovery)
2. offerte da parte dei servitori (offer)
3. scelta di una offerta (request)
4. conferma della offerta (pack)
5. messaggi prima della scadenza (lease)
6. rilascio dell'offerta (release)

NAT - *Network Address Translation*
trasla indirizzi intranet privati in indirizzi IP globali in rete aperta utilizzando le porte (in modo da usare un solo indirizzo esterno)

Azioni di gruppo: Broadcas e Multicast

Il broadcast in TCP/IP viene limitato nell'ambito della rete locale. Due tipi:

- broadcast limitato
  Diretto a tutti gli host della rete locale (tutti i bit di indirizzo a 1)
- broadcast diretto
  Diretto agli host di una rete specifica (netid identifica la rete, tutti i bit di hostid a 1)

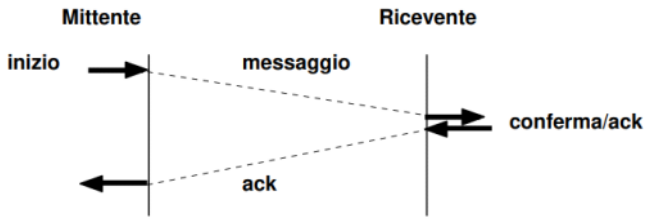
Il multicast è realizzato mediante gli indirizzamenti di classe D (4 bit più significativi = 1110) e tutti gli host che si sono registrati possono ricevere/inviare messaggi al gruppo multicast

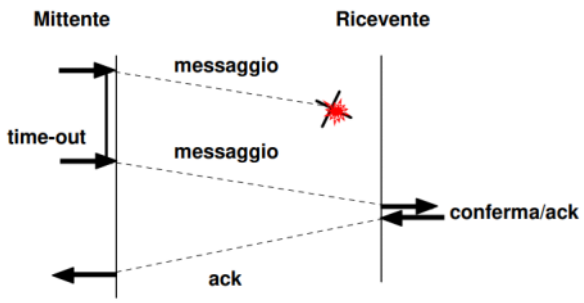
Comunicazione: ARQ (Automatic Repeat reQuest)

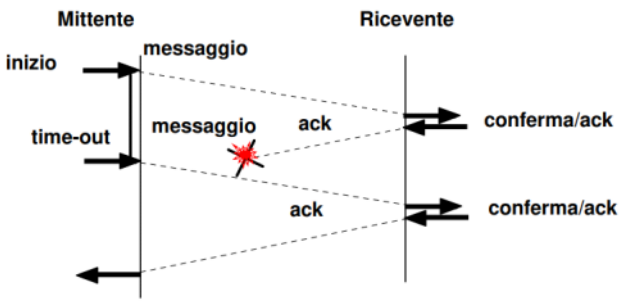
Per una comunicazione con distreta QoS occorre considerare due aspetti:

- **Affidabilità** - aspettare che il ricevente abbia ricevuto il messaggio
- **Asincronismo** - non aspettare troppo tempo

ARQ attua un meccanismo di stop and wait con ACK del messaggio:







Internet e protocolli

venerdì 13 dicembre 2019 10.59

Tipologie di ACK

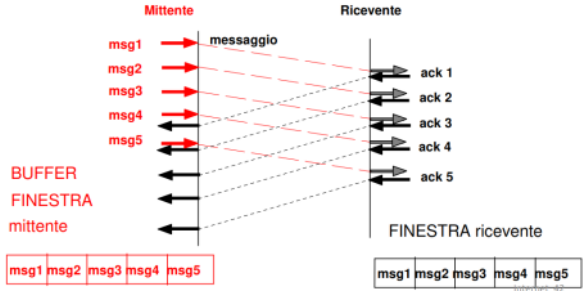
- ACK individuali (o selettivi) - Il significato è "ho ricevuto il frame n". Spetta dunque al mittente, nel caso un frame sia andato perso, fermare l'invio dei messaggi e ricominciare dall'ultimo frame ricevuto dal ricevente.
- ACK cumulativi - Il significato è "ho ricevuto tutti i frame fino a n". Lo svantaggio è che dicono poco sullo stato del ricevente: l'ACK indica sempre la stessa posizione nello stream ricevente anche se i successivi fossero arrivati
- ACK negativi - Il destinatario notifica la necessità di ritrasmissione di un singolo frame

Continuous requests

In questo caso non si attende in modo sincrono la ricezione della conferma (ACK) ma si mandano i messaggi in modo continuo:

1. Il mittente manda messaggi fino a saturare la finestra disponibile
2. Alla ricezione di ACK i relativi messaggi vengono scartati e viene fatta scorrere la finestra

L'attesa del mittente si ha così solo a finestra piena.
Ovviamente sarà il ricevente a dettare le regole sulla massima dimensione della finestra.



In caso di errore o messaggio non arrivato mentre altri messaggi successivi arrivati:
Go-Back-N - Il mittente continua a inviare i messaggi fino al riempimento della finestra, vengono eliminati dalla finestra i messaggi confermati da ACK mentre vengono ritrasmessi, in caso di time-out, tutti i frame da quello perso in poi.
Selective Retransmission - E' simile al Go-Back-N, con la differenza che si ha una finestra di ricezione oltre a quella di trasmissione. Selective Repeat accetta qualunque frame valido all'interno della finestra di ricezione. L'arrivo di frame corretti al ricevitore genera lo scorrimento della finestra di ricezione, l'arrivo di ACK al trasmettitore genera lo scorrimento della finestra di trasmissione. Se c'è una perdita, il protocollo si comporta come il protocollo Go-Back-N.

Sliding window

TX:

1. Il mittente numera ogni messaggio con NumSeq
2. Accetta una dimensione della finestra data dal ricevente TXWinSize
3. Mantiene il numero di sequenza dell'ultimo messaggio inviato NumLastMess e dell'ultimo ACK ricevuto NumLastAck

Obiettivo:
NumLastMess - NumLastAck + 1 <= TXWinSize
RX:

1. Decide una dimensione della finestra RXWinSize
2. Mantiene il numero di sequenza del prossimo messaggio atteso NumWaitedMess e dell'ultimo messaggio confermabile NumAcceptableMess

Obiettivo:
NumAcceptableMess - NumWaitedMess +1 <= RXWinSize

TCP

venerdì 13 dicembre 2019 12.21

TCP

- Stream **affidabile** e **full-duplex**
- Connessione o canale virtuale bidirezionale:
  La connessione end-to-end garantisce che il messaggio passi dalla memoria del mittente al destinatario con successo e che si mantenga il flusso costituito di dati in ordine preciso e non alterabile.

Nota: La connessione TCP NON impegna i nodi intermedi si usano solo le risorse dei nodi degli end-user

Il protocollo TCP si basa su alcuni principi e vincoli da rispettare:

- formato dei dati trasmessi (segmenti con header fissato)
- possibilità di dati urgenti
- regole per la bufferizzazione e l'invio degli acknowledgement (sliding window) e relativo formato
- possibilità di comporre messaggi e decomporre in segmenti
- meccanismi di de/multiplexing (vedi UDP) attraverso il concetto di porta per distinguere più processi su uno stesso host

Code Bit:

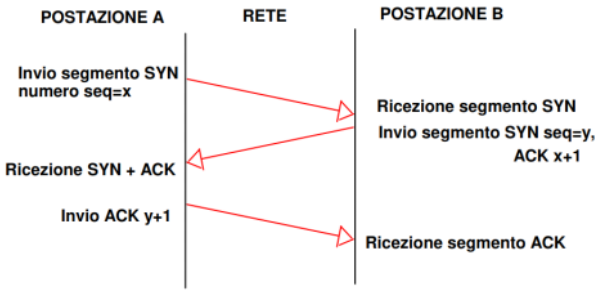
ACK	Il frame è un acknowledgment	RST	reset della connessione
URG	Il frame è urgente	SYN	stabilimento della connessione
PUSH	invio immediato del segmento	FIN	termine della connessione

TCP usa Go-Back-N ottimizzato con ACK cumulativi: si rimanda solo il primo, poi si aspetta per verificare se il ricevente manda ack cumulativo di una parte di flusso successiva. TCP fa anche uso di piggybacking per l'invio di ACK ossia, nel caso il destinatario debba inviare altri messaggi al mittente l'ACK viene incapsulato all'interno di essi per evitare di inviare troppo messaggi.

Fasi di operatività

Fase iniziale - three way handshaking.

1. A invia a B il segmento con SYN e richiede la connessione (con x: valore iniziale del flusso scelto da A)
2. B riceve il segmento SYN e ne invia uno identico ad A (con y: valore iniziale del flusso scelto da B) più un ACK con numero di sequenza x+1
3. A riceve il segmento SYN ed ACK e conferma la ricezione a B attraverso un ACK a sua volta con numero di sequenza y+1

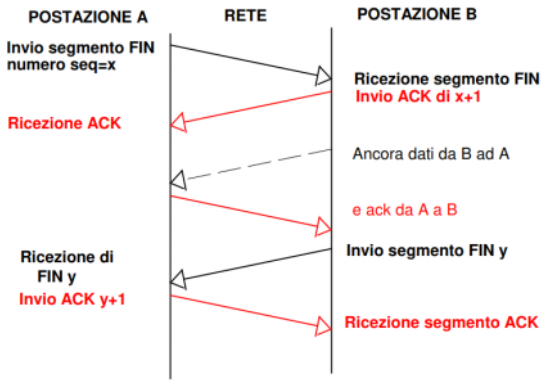


Negoziazione a tre fasi per stabilire se entrambi i nodi sono disponibili alla connessione. Avviene su modello di bidding (offerta senza rifiuto): ogni pari decide in modo unilaterale il proprio verso della connessione; il pari deve accettare.
In questa prima fase avviene anche il coordinamento sulla sequenza iniziale di valori:

- numeri di porta
- numeri di inizio per i flussi (x e y)
- tempo di trasmissione e risposta (time-out)
- finestra di ricezione

Fase finale - chiusura a 4 fasi

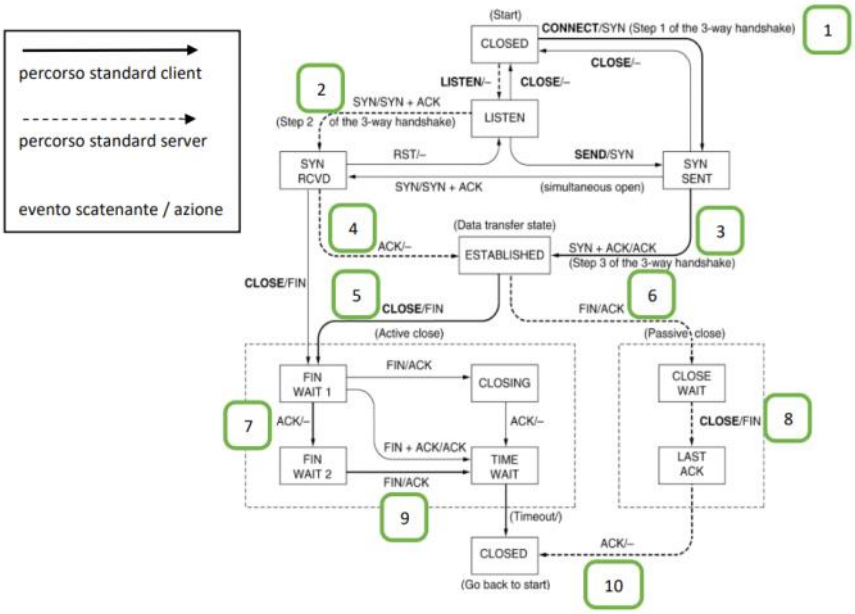
Si prevede una chiusura monodirezionale di output ossia definitiva per un solo verso (il verso di autorità) senza perdita dei messaggi in trasferimento e di quelli in arrivo. Se ad esempio A chiude nel suo verso di uscita allora A comunica a TCP di non avere ulteriori dati e chiude la comunicazione solo nel verso da A a B. I dati che precedono la fine sono ricevuti prima della fine dello stream da A a B.
Nota: per fare ciò TCP permette il passaggio sul canale intenzionalmente chiuso dei pacchetti ACK.



TCP

venerdì 13 dicembre 2019 12.53

Stati in TCP



Controllo di flusso

Il controllo di flusso è fondamentale in Internet in cui ci sono connessioni con macchine molto diverse fra loro.
Sono meccanismi fondamentali di coordinamento:

- la finestra
  La dimensione della finestra viene inviata per ogni segmento e comunica al pari quali siano le esigenze di memoria della connessione. Una finestra a 0 significa di non inviare alcune segmento. Ogni pari comunica all'altro la propria situazione con la finestra
- la dimensione preferenziale dei segmenti da inviare
  Attesa di dati prima di inviarli fino ad avere un segmento che sia conveniente inviare (Maximum Segment Size). Si deve evitare di avere trasmissioni di messaggi cortI (*Silly window*: in genere non si fanno azioni sotto una soglia e non si mandano finestre troppo piccole, così come non si mandano segmenti troppo corti)

Congestione

Intendiamo con congestione una situazione che può essere sia dipendente dai soli endpoint della connessione
stessa, sia da una più ampia situazione della intera rete come ad esempio tutti i router sono con buffer pieni e nessuno scambio può più avvenire, fino alla de-congestione oppure abbiamo forzato troppo il cammino della connessione e
abbiamo congestionato solo le nostre risorse locali.

Si risolve attraverso metodi di:

- **congestion avoidance** - Metodo di **slow start**
- **recovery** (dopo l'identificazione) - Per evitare di aggravare la situazione il mittente dimezza la finestra di invio e raddoppia il time-out

Slow Start

Lo Slow start è il transitorio sulla finestra del mittente per arrivare da una situazione iniziale fredda (senza comunicazione) ad una comunicazione a regime calda (diversa banda):
rwnd è il valore a regime, e ci si arriva partendo da finestre molto limitate che crescono in base all'assorbimento della rete e agli ack ricevuti con crescite differenziate:

- veloci inizialmente (sotto ssthreshold) - fase esponenziale
- limitate successivamente (sopra ssthreshold) - fase lineare

In caso di congestione presunta: dopo un timeout o ricezione di 3 ACK duplicati si ridimensiona tutto e si riparte in modo esplorativo: